ARPA ORDER NO. 189-1

MEMORANDUM
RM-6002-ARPA
SEPTEMBER 1969

THE GRAIL SYSTEM IMPLEMENTATION

T .O. Ellis, J. F. Heafner and W. L. Sibley

PREPARED FOR:
ADVANCED RESEARCH PROJECTS AGENCY

The RAND Corporation
SANTA MONICA • CALIFORNIA

MEMORANDUM
RM-6002-ARPA
SEPTEMBER 1969

THE GRAIL SYSTEM IMPLEMENTATION

T. O. Ellis, J. F. Heafner and W. L. Sibley

This research is supported by the Advanced Research Projects Agency under Contract No. DAHC15 67 C 0141. Views or conclusions contained in this study should not be interpreted as representing the official opinion or policy of ARPA.

The RAND Corporation
1700 MAIN ST • SANTA MONICA • CALIFORNIA • 90406

This study is presented as a competent treatment of the subject, worthy of publication. The Rand Corporation vouches for the quality of the research, without necessarily endorsing the opinions and conclusions of the authors.

Published by The RAND Corporation

PREFACE

This Memorandum is the third of a three-part[†] final report on the GRAIL (GRAphical Input Language) Project sponsored by the Advanced Research Projects Agency of the Department of Defense. The study was an integral part of both RAND's and the client's overall programs to explore man-machine communications.

The major problem in implementing this graphical programming system (GRAIL) was to provide good response times and simultaneously to relieve the man from rote system tasks. The Memorandum describes internal program representation and its dynamics along with algorithms employed to shorten the feedback loop.

[†]See also by the same authors: *The GRAIL Project: An Experiment in Man-Machine Communications*, The RAND Corporation, RM-5999-ARPA, September 1969; *The GRAIL Language and Operations*, The RAND Corporation, RM-6001-ARPA, September 1969.

SUMMARY

This Memorandum describes the major problems that were considered in implementing the GRAIL (GRAphical Input Language) system. The central issue was to shield the man from awareness of system operational tasks while providing good response times.

Interactive use of the console (a RAND Tablet/Stylus and a CRT display) demands that many independent data packages be accessed in real-time. The Memorandum examines the internal representation of the user's program and its dynamics. This representation consists of: 1) its picture form, 2) data structures to denote properties implied by the picture, and 3) positional information to relate stylus location to the other forms. Also discussed are the problems of storage allocation, data and storage management, and data transfer. Since primary storage is not large enough to contain the user's entire program nor the entire system, storage must be arranged and maintained so that the user is unaware of the size limitations.

As the man's highly variable actions are not predictable, the central processor must be allocated to meet peak demands asynchronously with respect to other tasks. This Memorandum examines the use of algorithms for scheduling, priority, synchronization, and parallel processing.

The system was tailored to provide good response times for operations on complex program organizations within limited time and space resources.

ACKNOWLEDGMENTS

The authors would like to thank the many reviewers of this Memorandum for their suggestions as well as the following people for their discussion of this work: G. F. Groner, R. Patrick, J. C. Shaw, and R. Turn.

CONTENTS

PREFACE iii

SUMMARY v

ACKNOWLEDGMENTS vii

FIGURES xi

Section
I. INTRODUCTION 1
User's Data Representation 1
Dynamics of User Data 3

II. PRIMARY STORAGE 5

III. SECONDARY STORAGE 9
Transfer Times 9
Formatting and Access 9
Data Set Mapping 10
Space Allocation 10

IV. DISPLAY DATA 13
Area Control Block 13
Pseudo-Channel Program Table 13
Channel Program Table 15

V. CODE PLANES 16

VI. RING STRUCTURE 17
Elements and Rings 17
Structure Space 19

VII. PRIMARY STORAGE DYNAMICS 21
Read-Only Process Space 21
Context Space 22
Automatic Space 22
Central Processor Allocation and Supervisory Functions 23
Parallel Processing 23
Task List 24
Serially Reusable Process (SRP) Scheduling 24
Program Status Group (PSG) Synchronization 25

VIII. DISCUSSION 26

Appendix
A. BASIC RING STRUCTURES 27
B. LOADING AND THE CORE CONTROL DICTIONARY .. 46

REFERENCES 55

FIGURES

1. System Organization 6
2. Space Allocation 7
3. Head Format 11
4. Display Control 14
5. Ring Structure 18
6. Space Allocation Structure 28
7. Available Heads Element 29
8. System Structure 31
9. Files Description Structure 32
10. File Structure 34
11. Context Structure 36
12. Context Structure 37
13. Plane Structure 40
14. Plane Structure 41
15. Plane Structure 42
16. Plane Structure 43
17. Plane Structure 44
18. Plane Structure 45

I. INTRODUCTION

This Memorandum, the third of three (see Refs. 1 and 2), describes some aspects of implementing the GRAIL system. The software was planned and coded according to the GRAIL philosophy, yet tempered by hardware† specifics. The closely-knit data organization and management were tuned to the hardware to provide timely, expected responses. The total organization consists of many independent data packages--some pictorial, some processes, some structural--all of which must be accessed in real time.

Since the system was written within the GRAIL language conventions, processes described and compiled from the console are identical in form to hand-coded system processes.

USER'S DATA REPRESENTATION

Internal representation of the man's construction-time program is a central issue to the utility of the system. The following requirements should be satisfied:

1) Data organization and management should be carefully matched to the man's response expectations; e.g., no hesitation in inking or symbol recognition should occur, but major changes in the operating environment could be fed back at a more leisurely pace.
2) The man should feel that he is working directly with his problem, thus changes in representation should be made automatically only when his attention is focused on the area to be modified. Therefore, the man should be able to retrieve exactly the picture he constructed rather than some logical equivalent.

---

†The hardware was an IBM System/360 Model 40, two IBM 2311 Disk Drives, A Burroughs Corp. prototype display, and a RAND Tablet with match circuitry and direct feedback.

3) The representation should be easily amenable to such different kinds of processing as querying, editing, or compiling. The representation chosen is associative.

The internal representation chosen to satisfy these three requirements consist of the following three parts:

1) The display order codes generate the picture exactly as it was organized by the man.
2) The data ring structure (a partial abstraction of the pictorial representation) complements this by containing the implied properties of the picture--i.e., the connectivity of both control and data.
3) A set of linked tables connect the picture to the structure and correlate with both the geometric areas sensitive to the stylus and the stylus position itself.

The display picture segments are highly dynamic because the man frequently modifies them. They are represented so that it is easy to detect the display area that is being stylus-addressed and to alter the display easily in parts. The linked tables may be considered a plugboard to which display segments or picture elements can be added or deleted. With a single pen motion the man can operate on a flowchart symbol or code statement, a collection of symbols or statements, or the entire display. The plugboard must be addressable at these various levels and, at the same time, be insensitive to the number and displayed arrangement of segments. The linked tables contain the channel program for driving the display picture, links to the corresponding ring structure, and coordinates of picture elements for area comparisons with stylus location.

The hierarchy of the language--viz., files, closed processes, open processes, and frames--suggests some form

of structural representation other than an array. Furthermore, in using the language the man is building and changing such associations as the connectivity of a flowchart--which suggests even more strongly a structured (open-ended) form for manipulation.

The designers chose a two-dimensional ring structure with forward pointers. Unidirectional pointers conserve space and can be justified if the number of elements of each ring is small enough so that a particular element can be found quickly; e.g., a flowchart symbol generally has few attributes. Eliminating the background "garbage collection" common to many list processors is also desirable from the standpoint of response time. Two element sizes (assignable from opposite ends of available space) allow immediate compression of available space; thus no space-compression overhead occurs when one structure is exchanged with another on secondary storage; e.g., when the man goes from picture to picture. Since the picture description is in display language, only the cross references between logical structure and display elements need be kept in the structure.

DYNAMICS OF USER DATA

Since primary storage is not necessarily large enough to contain the man's entire file, storage is arranged and maintained so that he is unaware of size limitations. Information subject to immediate operations is kept in primary storage--i.e., the currently displayed picture and related ring structures; other pictures and structures are retrieved from secondary storage as needed. Responses correspond to his expectations since manipulations within a picture require no secondary-storage access, and picture-to-picture operations within a plane require only frame swapping.

Specifically, the data kept in primary storage are the current display frame, a ring structure abstracting the current plane connectivity, a ring structure abstracting

the open processes and labels within the current closed process, and a ring structure denoting the closed process relationships within the file.

II. PRIMARY STORAGE

Primary storage contains systems and user (the man's) data sets--processes, structures, and displays (see Fig. 1). All the data sets that the system needs to supervise the man's actions cannot reside in primary storage simultaneously. The highly variable demands for both data and their managing processes require dynamic allocation of primary storage that is automatically provided by the system. Primary storage is partitioned into three segments--*read only* (R/O), *context*, and *automatic* (see Fig. 1)--which accommodate R/O processes, formal parameter linkage, and temporary storage. Once loaded, read-only processes occupy the space until it is needed. The parameter link and temporary space are assigned only for the execution of an instance of a process. Only the management of automatic is described in Fig. 2, since all three space types are managed similarly.

The supervisor (one of the basic system processes) keeps the automatic segment of primary as linked blocks of available space and space that has been assigned to processes being executed. Each block contains forward and backward links followed by the space addressable by the active process and perhaps subsequent daughter processes.

Automatic space is conventionally allocated from lower to higher core and normally occurs in large blocks. The sequential-in-time nature of process calls and the release of space upon their returns (in inverted order) help to accumulate large blocks. Exceptions occur when processes are invoked late in a sequence of actions, yet have a long active time compared to processes invoked even later. Many of the processes with parallel exits have this latent release characteristic. When such processes are compiled, they are given a distinct data-set type that the supervisor recognizes when allocating automatic space. The space is assigned from higher to lower core, thereby maintaining

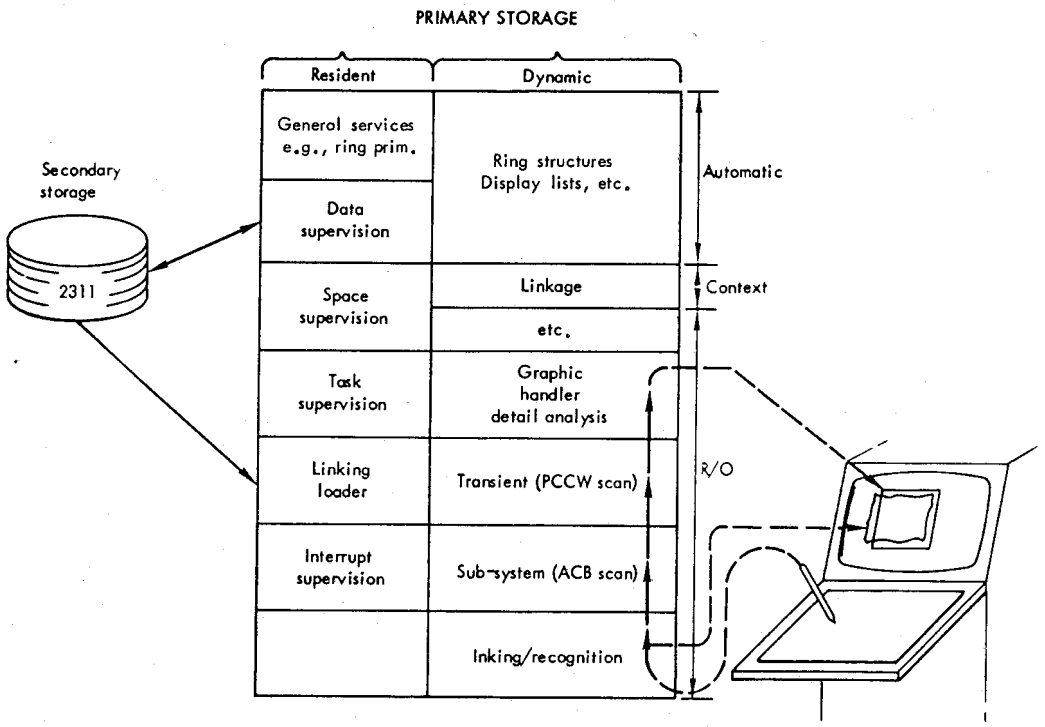


Fig. 1--System Organization

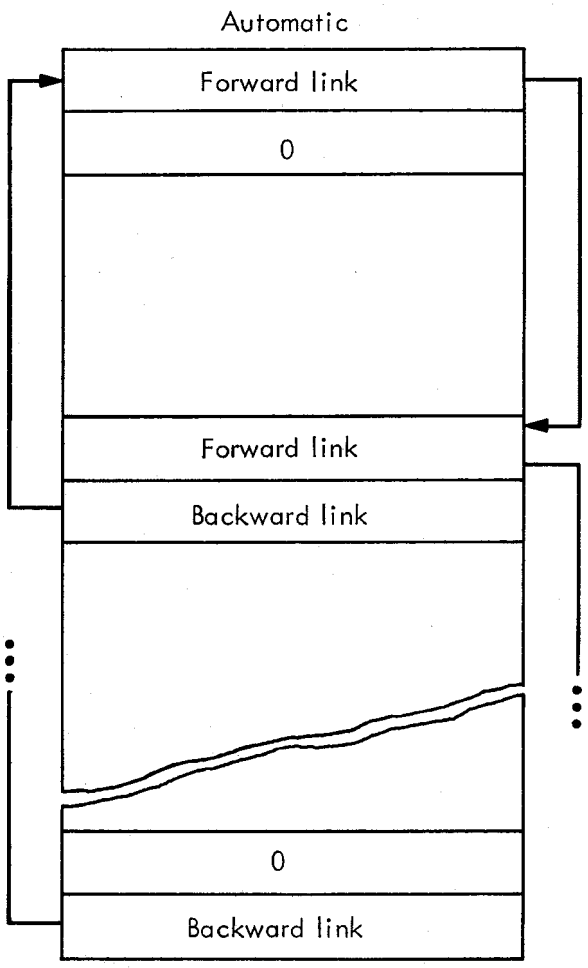


Fig. 2--Space Allocation

large available blocks in the center. No contiguous blocks of available space occur since the return algorithm examines blocks adjoining the block being released and merges those available.

III. SECONDARY STORAGE

Secondary storage houses system data sets (read-only processes, fixed display frames, and ring structures) and user-file data sets (structures, frames, and compiled processes). All data sets conform to a common mapping on secondary storage, generalizing the I/O processes that manipulate them. The space is assigned dynamically because its contents are as highly variable as those in primary storage. Standardized physical formats minimize transfer times and simplify channel programming.

TRANSFER TIMES

Two IBM 2311 Disk Storage Drives house the data sets. For transmission the read/write heads must be positioned to the desired cylinder and head (the address of a disk data space), which takes an average deflection time of 180 ms[†] over 200 cylinders. The system uses 14 adjacent cylinders on one device and a given user's file employs 8 adjacent cylinders on the other, reducing the average deflection time. For many manipulations within a file that require disk access, the deflection time is zero.

A second delay time averaging 12 1/2 ms (25 ms/revolution) occurs before the disk revolves to head origin; then 25 ms are required to transmit the data.

The average transmission time for additional heads of the same data set is less than two revolutions, since the space-allocation algorithm tends to assign heads sequentially.

FORMATTING AND ACCESS

All heads are formatted with a single physical record (R1) of maximum length to simplify both channel programming

[†]Engineering changes have reduced this figure.

and CPU processing. The transmission times for storing and retrieving a data set with this format are effectively the same as using multiple physical record markers, unless the existing record format of the head is unknown. The description for random formats must be maintained in primary storage or a lengthy channel program must be executed to ascertain them.

DATA SET MAPPING

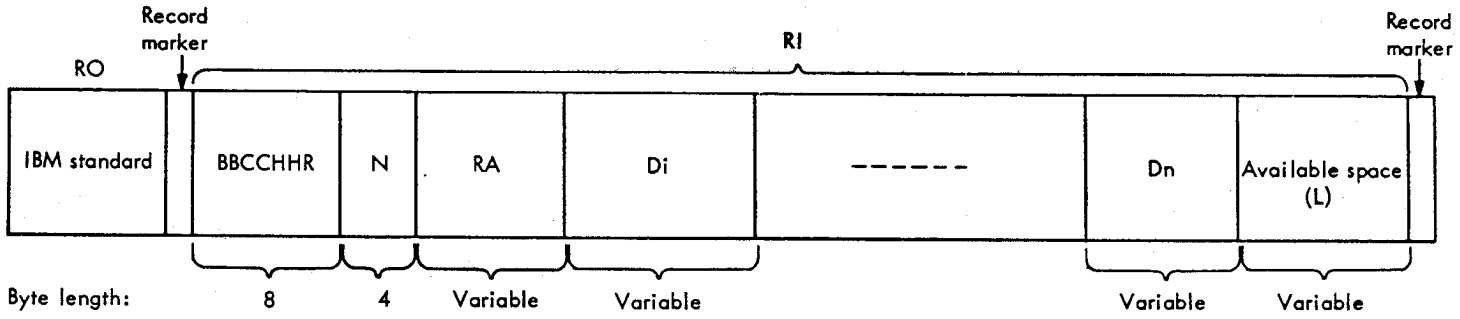
The same logical I/O group of processes manages all data sets. Because they differ radically in their internal organization, a header is prefixed to each data set to form a logical record (see Fig. 3). Both the header format and the logical I/O processes are open ended for defining new data-set types. The header for each data-set type is uniquely numbered and contains the cylinder-head of the next segment of the data set (if it spans more than a single head). The remaining header items are different for each type.

SPACE ALLOCATION

The logical I/O process group assigns data-set space upon output request. The old version of a data set is deleted from secondary storage when the new version is written, rather than when the old version is read, thus providing better error recovery. Since space is assigned dynamically, a data set is arbitrarily located each time. Parent processes of the logical I/O group may keep relative addresses. The I/O processes compute absolute addresses for channel program use only.

A ring-structure data set at a fixed location describes the space allocation and data-set locations currently on each disk pack.

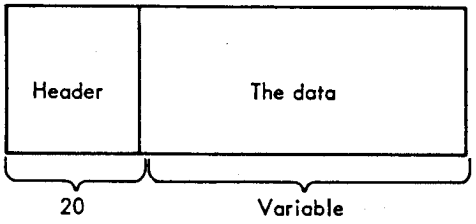
The space-allocation algorithm computes the amount of space required for the data set. If the amount required is



N = Number of data set entries this head

RA = Addresses relative to R1 of each logical record

Di = A logical record



L = Current unused space - also reflected in the ring structure describing this head

Fig. 3--Head Format

greater than a full head, a full head is used and the remainder is again tested against full-head count. A remainder less than a full head is checked against a partially available-heads structure. If enough partially available space is found, the data-set remainder will be assigned there; otherwise, the remainder is assigned to part of a full head. When a data set requires several full heads, they are normally contiguous.

IV. DISPLAY DATA

The display data consists of:

1) The display order codes that cause the CRT to display a picture;
2) A continuously looping collection of channel program segments (CCWs) that drive the CRT;
3) Pseudo-channel program segments (PCCWs) that relate display orders to structure and both to pen-sensitive areas.

AREA CONTROL BLOCK

The CRT surface is dynamically divided into many virtual areas. The area control block (ACB) relates large display areas to their channel programs, to the stylus, and to processes that interpret stylus motions (see Fig. 4). A high-level process (subsystem), active for all the current areas, directs stylus data to a lower-level process (transient) that monitors the particular area. The ACB links the stylus, the CRT, and the transient process-group that interprets pen motions and responds to them in that display sub-area.

The x,y coordinates in each entry of the ACB (Fig. 4) define a rectangular area of the display surface in which some display information appears. The PCCW entry that points to a table associated with that area is also addressed. The type byte identifies the kind of information (and its transient process-group) displayed on that part of the CRT--e.g., flowchart or code sheet--and indicates its status--i.e., primary or secondary (foreground-background).

PSEUDO-CHANNEL PROGRAM TABLE

A variable-length PCCW table for each ACB entry describes in more detail the contents of that display area. A PCCW table consists of a header to address the first and last CCWs

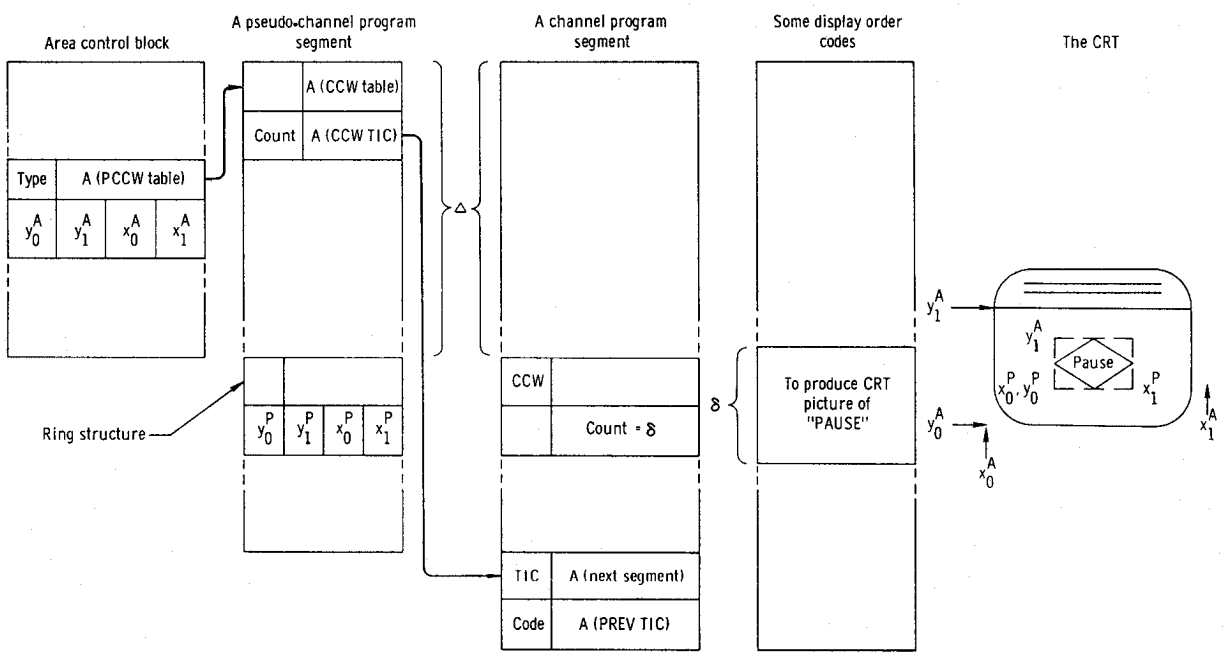


Fig. 4--Display Control

of the corresponding CCW table and of a count of the remaining entries in the PCCW table that correspond positionally to complementary CCWs in the CCW table. A PCCW entry occurs, for example, for each flowchart symbol displayed in the area bounded by the ACB coordinates, for each virtual button, or for each code statement.

The second half of the PCCW entry defines a virtual rectangle surrounding the displayed element, while the first half is meaningful only for flowchart symbols, coupling the figure with the ring structure describing such attributes as labels and connectivity.

CHANNEL PROGRAM TABLE

A CCW table (for each ACB entry/PCCW table) controls the CRT order codes for the ACB display area. The table contains a header pointing to the fixed display in the ACB-described area--e.g., the grid lines and title information for a code sheet; a CCW for each flowchart symbol or code line; and a transfer-in-channel (TIC) command to the next CCW table segment for the next ACB-described area.

The first word of the TIC is standard for channel operation. The second word locates the next and previous segments for purposes of linking a segment in or out. It has a code that is the ACB code for the entry corresponding to the next segment. The address field points to the TIC of the previous segment.

V. CODE PLANES

The man may define processes in assembler language as well as by flow diagrams. A code plane in primary storage when being displayed is organized similarly to a flow diagram--each CCW addresses the CRT orders for a single coding statement. Only the CRT orders are recorded on secondary storage, since the channel program is constructed for a code plane and represents only that part displayed before the *viewing window* at any one time.

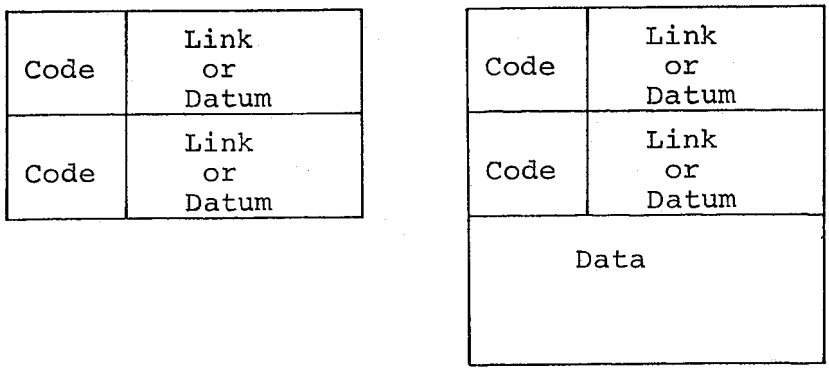
When code statements are syntax-analyzed at end-of-message, only label information is abstracted to the ring structures; the original CRT picture form is kept intact.

VI. RING STRUCTURE

Ring structures describe parts of flow-diagram pictures and disk-space allocation on both the system and the user's packs.

ELEMENTS AND RINGS

Eight and 16-byte sizes were used. Each 4-byte word of an 8-byte element and the first two words of a 16-byte element consist of a code and either a link or a datum. The next two words of a 16-byte element are data, as shown below.



One bit of the code byte determines (the byte has further application) if the three remaining bytes constitute a link or a datum. Links are never examined by a high-level process using the structure but only by a group of ring-structure primitives, written as remote code sequences. All links are relative to the base of the space in which the ring structure resides.

The upper link is referred to as the *object* link and the lower as the *set* link (Fig. 5). One or more elements linked through the object link are known as an *object ring* and, similarly, one or more elements linked through the set link constitute a *set ring*.

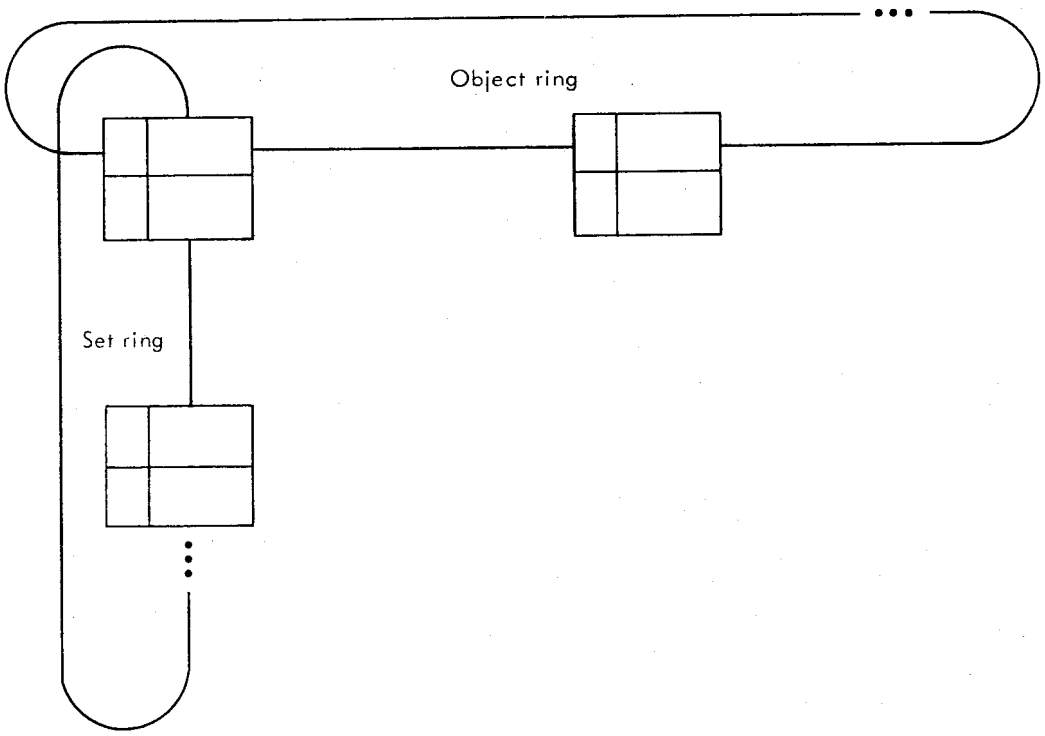
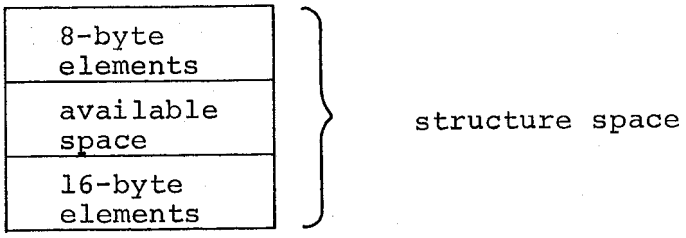


Fig. 5--Ring Structure

STRUCTURE SPACE

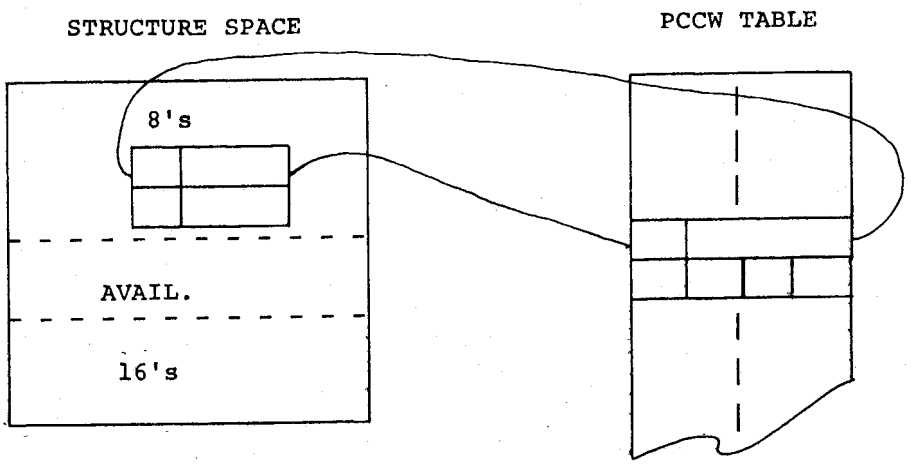
Rings of elements are spatially compartmented in a *structure space* (shown below).



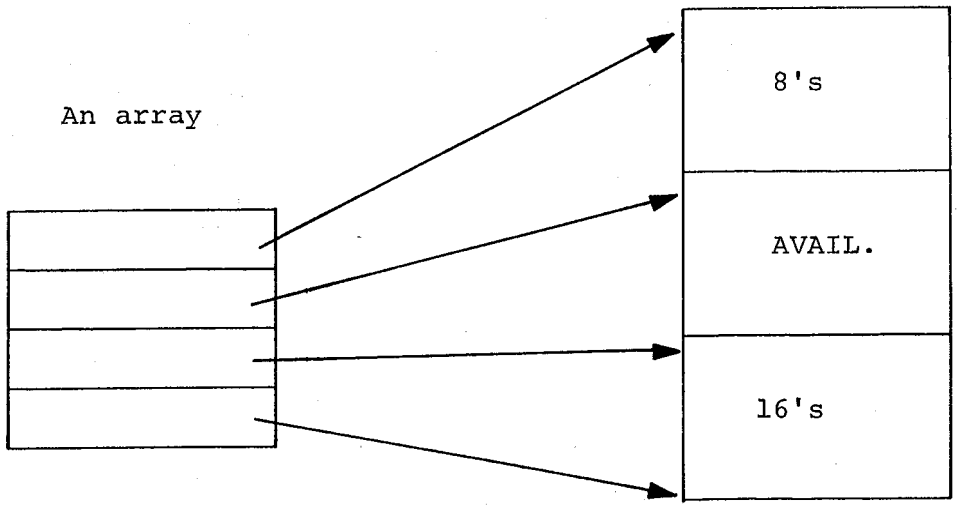
The boundaries of available space change according to the number of elements in use at any given time. When either length element is requested, that boundary element is supplied; and when either is returned, the boundary element of that type in use is interchanged with the element to be returned. Thus the used elements stay compressed in their respective groups and available space remains contiguous.

To conserve secondary-storage space, only the elements in use are recorded.

Elements external to the structure space are permitted while the structure is in primary storage. An external element gives processes an easily accessible "handle" to a ring--e.g., the PCCW (shown below).



The allocation of a structure space is controlled by a block of pointers, external to the space, called an *array* (shown below).



The pointing is normally maintained by the primitives, not by a higher-level process; but the array usually resides in the same automatic store as the structure space for some high-level process instance.

VII. PRIMARY STORAGE DYNAMICS

READ-ONLY PROCESS SPACE

The read-only space (R/O) houses processes that are loaded when an instance of the process is encountered at execution time. The supervisor invokes the loader to retrieve the process from secondary store, allocate space for it, and resolve references. Functionally related processes are usually loaded as process groups. The loading operation is implicit and does not appear as part of a flow-diagram description. It is executed as a parallel task to allow data transmission to overlap other activities.

Assembler output is preprocessed before being catalogued as an R/O data set on secondary storage, but it is not formatted as an absolute load module. The data sets are relocatable and are not necessarily loaded at the same time as other processes referencing them. A core-control dictionary (CCD) resides in primary storage to resolve references among loaded processes. A process-group data-set contains:

1) A compressed section for resolving references local to the group;
2) A compressed external-reference section for resolving references from the group to processes not included among them;
3) A definition section for resolving references into the group from other external processes;
4) The read-only process definitions themselves.

References (and access) from one process to another are made only through the entry point, as defined by the flow-chart symbolism.

A process group remains in primary after execution until the space is needed for other processes. When space is insufficient to load another group, all dormant process groups (those not in execution) are released (deloaded) from

R/O space. If completion of a task is pending because of an interrupt, a WAIT synchronizer, or a pending return from a daughter process, the space is not released. Additional rules cause process deloading as a function of the contextual level of the process being invoked.

The preprocessed data sets, the CCD, and the loading algorithms were organized to optimize response times.

CONTEXT SPACE

The contents of the *context* space describe the operating environment of a unique use of a process at execution time. Context space is automatically allocated at execution time for each use of a closed (labeled) process or a serially reusable process. The space contains parent- and daughter-process context linkage information, translated parameter pointers, and return location information.

The supervisor assigns space when an instance is encountered and releases it when the invoked process returns control to the parent (invoking) process.

The parameter pointers address the real data at object time that the man indicated at construction time on the instance's Translation Frame.

AUTOMATIC SPACE

The supervisor assigns and releases *automatic* (auto) space along with context space. Auto contains the local temporary data for each instance of a closed process during its execution. The local real data at execution time corresponds to an auto specification that the man described on the Parameter Frame at construction time.

The ring structure and the display frame description of a man's construction-time processes are examples of temporary data for some of the system processes.

CENTRAL PROCESSOR ALLOCATION AND SUPERVISORY FUNCTIONS

The system interprets and evaluates stylus input. The tablet channel program is continuously looping and command chained. When the stylus switch is closed, the hardware inputs data at a rate of 4 ms/x,y coordinate. An interrupt occurs every 28 ms; i.e., every seven samples. The CPU speed and the amount of processing required for immediate inking feedback and stylus data analysis determines the channel program's buffer length. When the stylus is closed and moving slowly, it requires about 35 percent of the CPU. At maximum slew rate, Mod 40 speeds will not quite maintain the inking function, which generates fixed, small incremental coordinates from the variable incremental raw pen data for display.

A selector channel supplies data out of primary storage to the display hardware. A looping, continuous channel program, consisting of the linked table CCWs addressing the display order codes (described earlier for picture elements) drives it. The primary storage buffer is desirable for such highly dynamic data as the ink track.

PARALLEL PROCESSING

The system processes tablet input at a high priority. Ink and other responses are given on the CRT. Display frames, associated structures, and their managing processes are stored and can be retrieved from disk on demand. Supervisory functions permit logical I/O processes and process loading to overlap secondary-storage transmission with pen analysis and inking feedback.

The GRAIL language (and underlying supervisor) conceptually permits the simultaneous execution of two or more tasks that are (over some interval) independent--i.e., multiprogramming; and given the requisite hardware it would permit multiprocessing.

The treatment of stylus inking feedback on the display surface exemplifies the parallel processing capability. An inking process is initiated as a continuous high-priority task. Other tasks may load processes from secondary storage or interpret pen data independent of the ink task. Tablet input independently invokes the ink task from a *wait* state.

TASK LIST

The task list (actually a table) holds tasks that may be dispatched (executed). The priority of task firing is last-in/first-out--under the assumption that the most recently suspended activity is the most important. The supervisor maintains this table.

The following cause dispatchable tasks to be entered on the task list:

1) Hardware interrupts stack the mainline interrupted task;
2) Parallel exits stack the initiating task;
3) Setting a waiting synchronizer stacks the continuation from the WAIT;
4) The terminal exit from a serially reusable process (SRP) with a pending use stacks the return task.

SERIALLY REUSABLE PROCESS (SRP) SCHEDULING

The supervisor schedules SRPs on a device basis after assignment of *context* and automatic space. If the SRP definition is executing another use, the pending uses are linked via their context space to form a queue. They are not placed on the task list since they are not yet dispatchable.

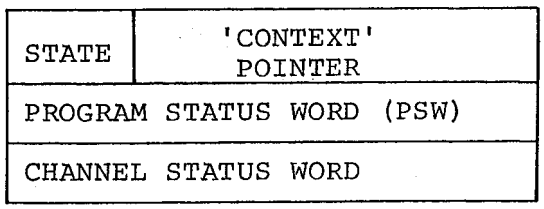
When an SRP terminally exits, the supervisor normally returns control to the parent process after releasing context and automatic space. If another use of the SRP is

pending, the parent return is stacked on the task list and the pending use of the SRP is initiated.

The priority of pending uses is first-in/first-out.

PROGRAM STATUS GROUP (PSG) SYNCHRONIZATION

The format of a PSG referenced by the flowchart symbols WAIT and SET is shown below.



The *state* byte indicates what state (e.g., WAIT or SET) the PSG is in. The *context pointer* addresses the context space in which the WAIT occurred. The PSW contains the process state and the instruction-counter value of the instruction subsequent to the WAIT.

VIII. DISCUSSION

GRAIL exemplifies a small operating system designed for one specific purpose. However, it has several characteristics in common with those found by other investigators. The size, complexity, and proliferation of subroutines indicate the need for dynamic storage allocation and demand loading and releasing of processes. No natural place to hide system processing seems to exist. The fast-response requirements coupled with the asynchronous and somewhat unpredictable behavior of the man make priority-driven multiprogramming imperative. Aside from the mechanics, the techniques developed are applicable to a large variety of problems represented by block diagrams.

Appendix A

BASIC RING STRUCTURES

Five ring-structure types are basic to system maintenance and representation of users' files. The purpose of each is described below with examples of its use.

SPACE ALLOCATION STRUCTURE

The space-allocation-structure complex (Fig. 6) is part of three major structures that describe the occupied and available space on either part or all of the secondary-storage disk pack.

The base element specifies the cylinder, head (C, H) of the beginning of a contiguous block of space. The occupied set gives the C, H relative to that base for data sets along with the data set ID. Not all data sets in the space appear on the occupied set. This complex is used by a group of logical I/O processes, and they either seek and append occupied elements or pass the relative C, H to the parent process for recording elsewhere. Typically, static display forms and compiled read-only processes are kept on the occupied set and dynamic display frames and ring structures are not.

The partially available object-ring elements specify relative addresses and byte counts for the available space on heads that contain data sets and for some available space as a block at the end of the head. This structure is also maintained by the logical I/O processes.

The available heads object specifies the availability or nonavailability of each head within the space. Figure 7 shows the format of an available-heads element. The space-allocation algorithm assigns contiguous heads for data sets requiring more than one head, except possibly for the last segment of the data set that may be assigned to any one of

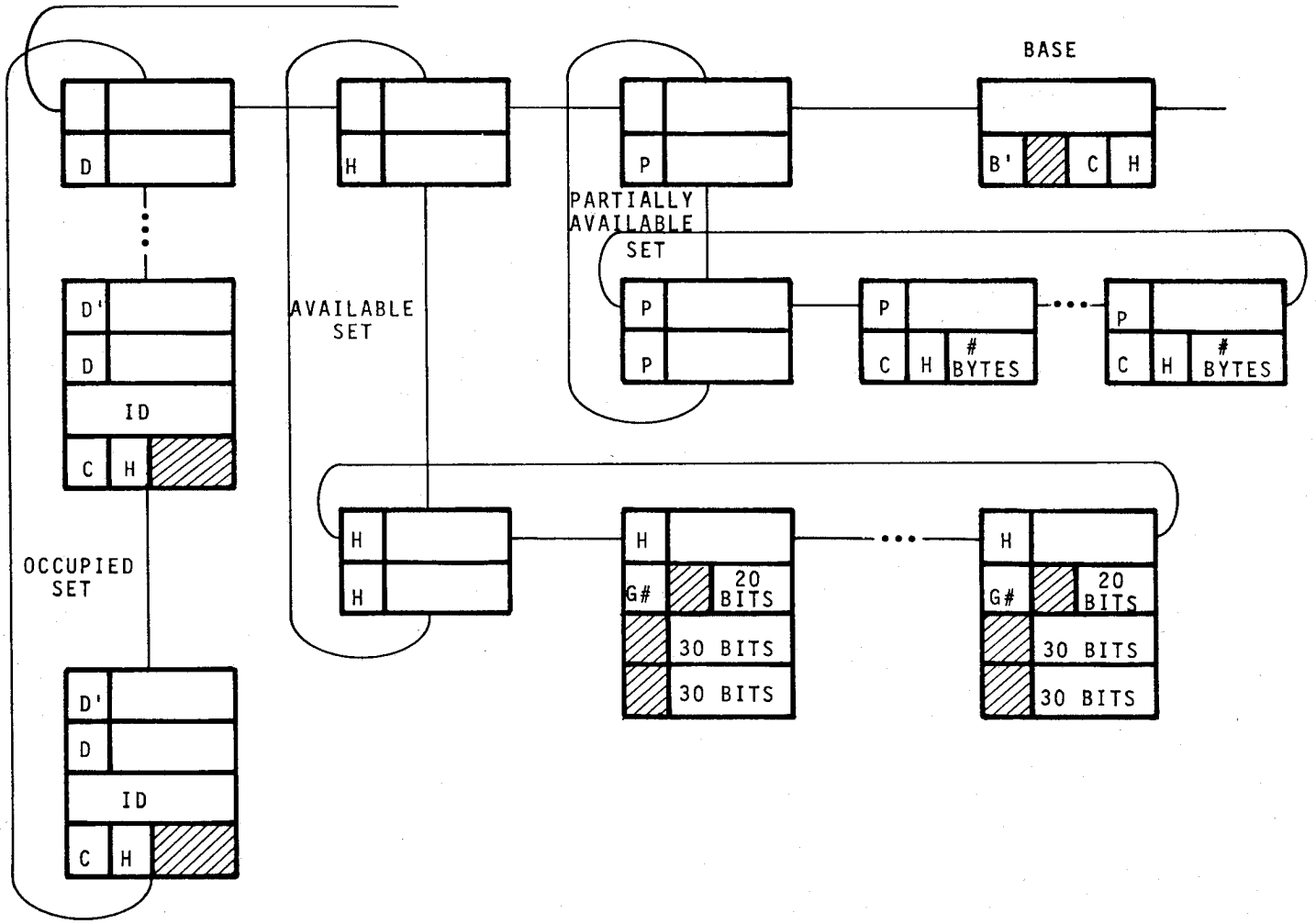
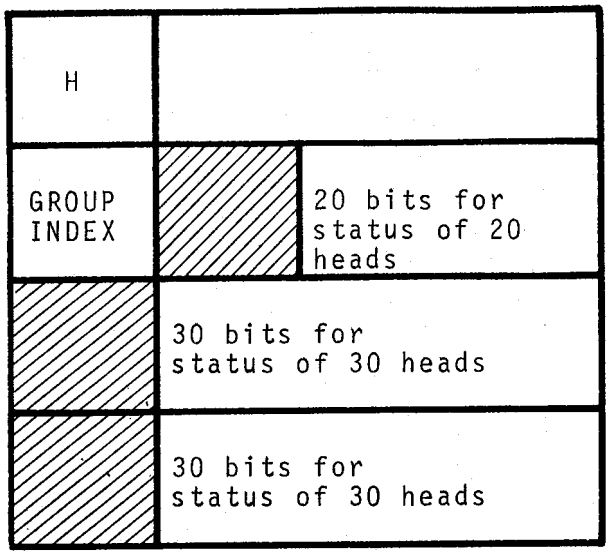
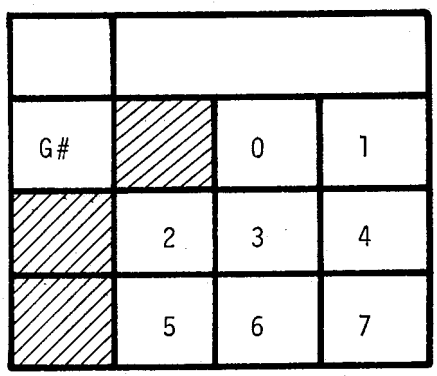
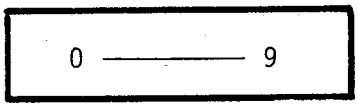


Fig. 6--Space Allocation Structure





Cylinder Nos. relative to the group index



Head nos. within a cylinder (1 bit = avail., 0 bit = partially or occupied)

Absolute C,H = (Base C,H) + (Rel. C,H)

Rel. C,H = (G# * 8) + (Rel. cyl. No.) + (Rel. Head No.)

Fig. 7--Available Heads Element

the partially available-heads space. Read-only compiled processes and occasionally code planes span more than a single head; ring structures and other display frames do not.

This ring is also managed by the logical I/O processes.

SYSTEM STRUCTURE

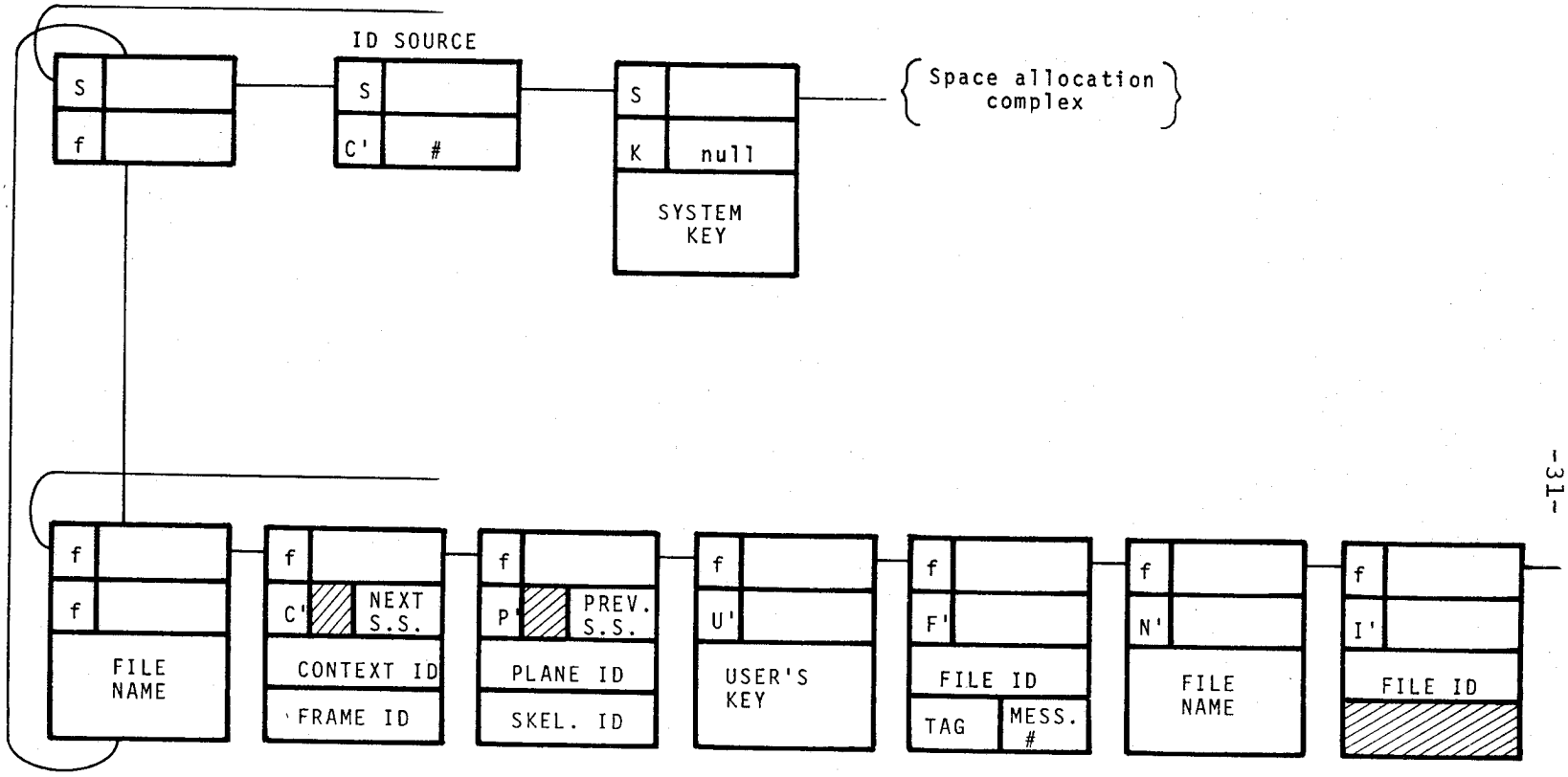
The system structure (Fig. 8) describes space allocation on the GRAIL system pack and also contains information about the current or last-addressed file. It is a bootstrap directory and (excepting the Initial Program Load record) the only data set that resides at a fixed location on the system pack. When GRAIL is in use, a copy resides in primary automatic storage of a basic system process that is always in use. The copy is rewritten on secondary each time the operating environment changes; i.e., a new subsystem is invoked.

The ID source element contents are passed as a formal parameter throughout the system and used as a source for internal identifiers of new data sets. The system key is a label identifying the current version of the system. Details of the space-allocation complex were shown in Fig. 6 (p. 28). The complex describes the entire system pack, which includes system read-only processes and invariant display frames.

Information about the current file contains internal identifiers of the structures and display frame currently in primary.

FILES DESCRIPTION STRUCTURE

The files description structure (Fig. 9) describes space allocation and user-file information on a user pack. It is a bootstrap for the user pack and is the only data set residing at a fixed location on the pack. A copy is



N' and I' apply to the current file when it is being executed, at which time f, F' are used for a pseudo-process parent to the file being executed.

Fig. 8--System Structure

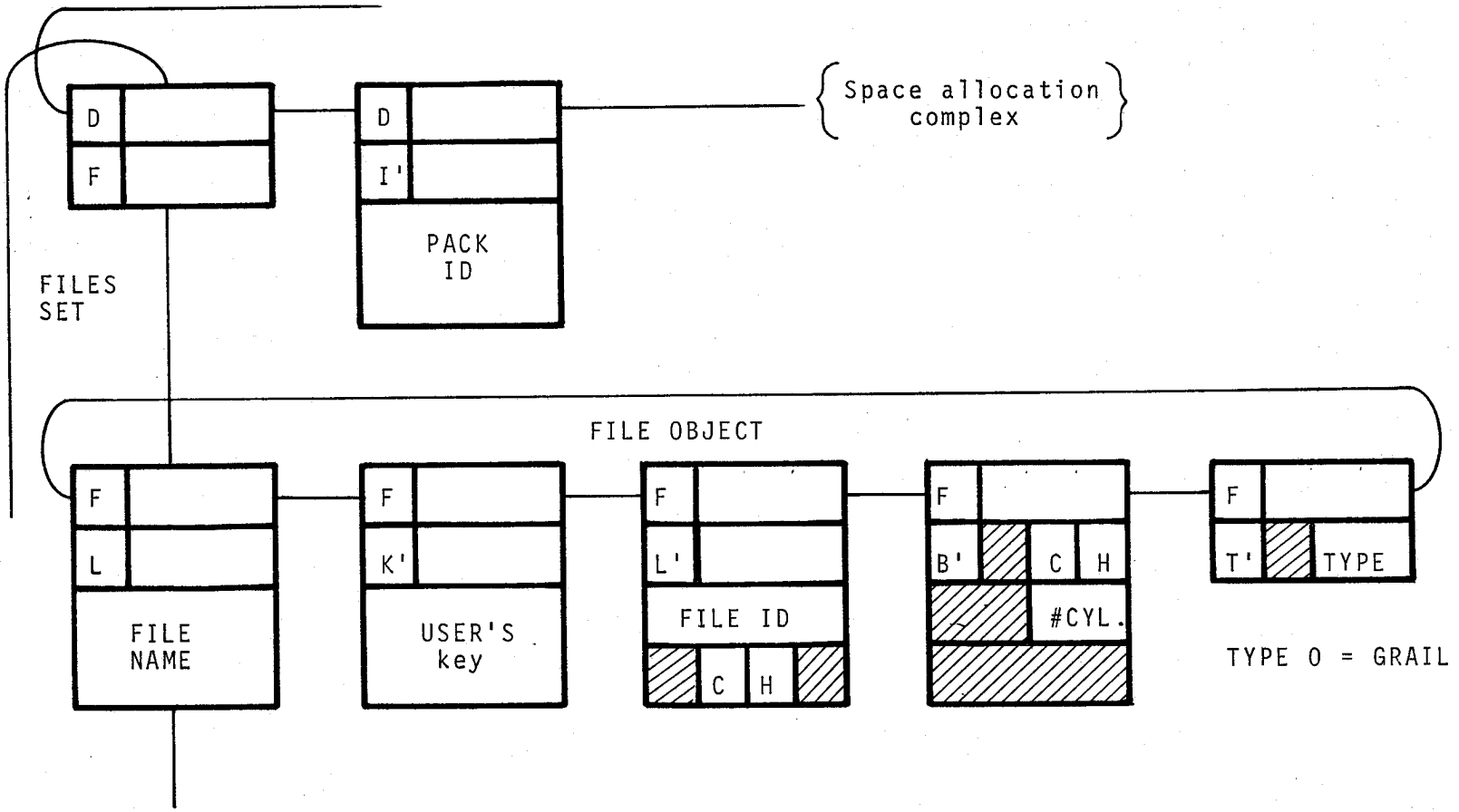


Fig. 9--Files Description Structure

not kept in primary as with the system structure, but it is read when needed to perform operations on entire files.

The structure contains a pack identifier, the space-allocation structure for the entire pack, and a set of file object rings (one for each file on the pack). The attributes of a file object are: 1) the file name, 2) the user's identification, specified when logging-on, 3) the file structure data set ID and relative location, 4) the file-structure-space (user's file) base address and extent, and 5) a type identifier. The type is necessary at this structure's level of use because other GRAIL derivative programs are supported in addition to the flowchart language.

FILE STRUCTURE

The one file structure (Fig. 10) in each user's file space describes the space allocations within the file space on the user pack and also specifies the interrelationship and the location of labeled-process definitions within the file.

One cylinder of the file space serves as a virtual memory during interpretive execution of processes in the file. The space allocation describes space within the file space. The occupied disk set contains ID and location only for compiled process data sets within the file. The modification number is a source within the file of identifying accesses to secondary storage, display-frame data sets, and some ring structures.

The *irresponsible-contexts* set links those labeled-process definition object rings that have a definition but no instances. Labels and contexts link all labeled-process-definition object rings within the file. The attributes of a context object are its ID and location, its name, the ID of the process from which it was last referenced, the process kind (e.g., serially reusable process), its type definition (code or structure), and its uses and used-by structure.

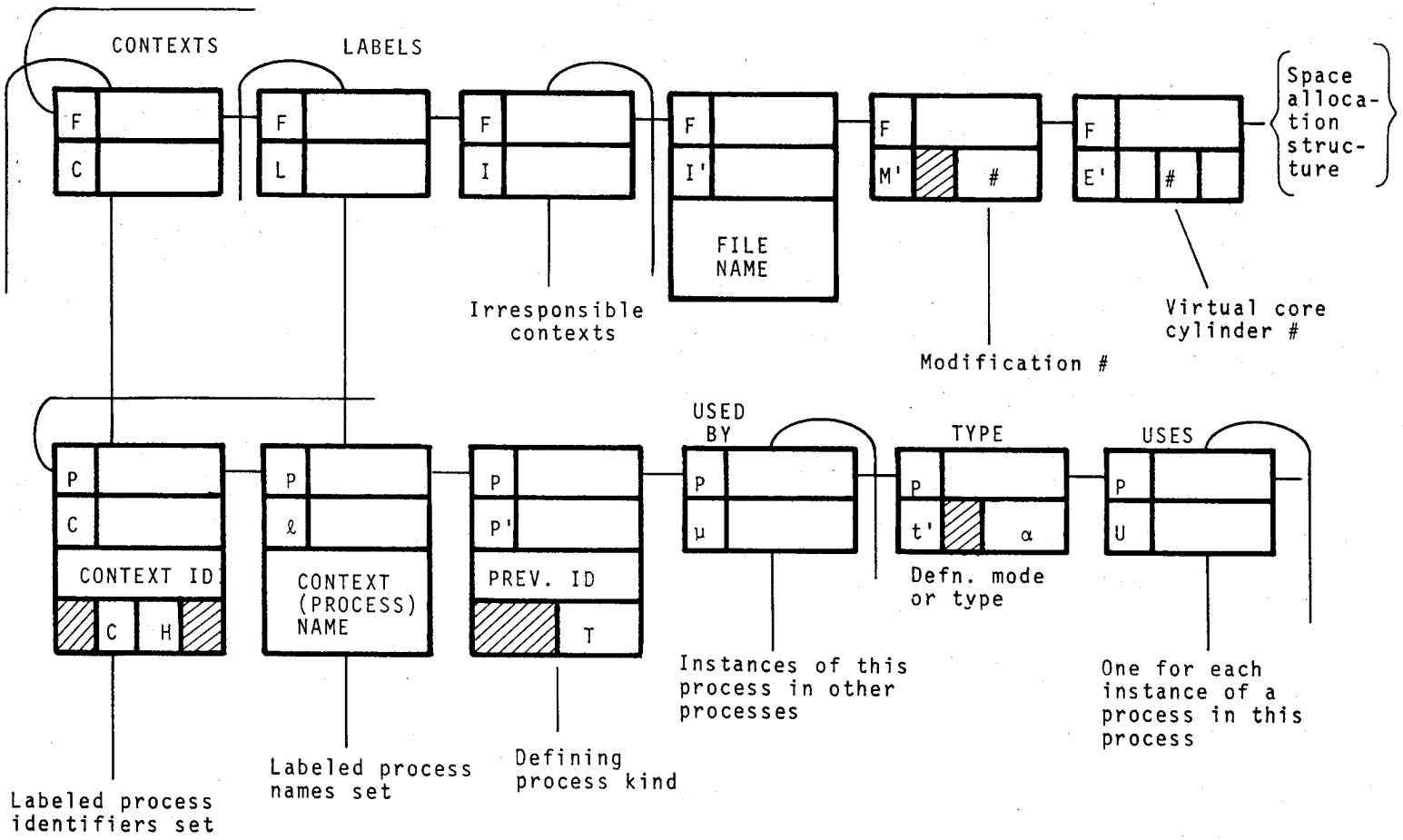


Fig. 10--File Structure

CONTEXT STRUCTURE

A context structure (Figs. 11 and 12), which exists for each labeled-process definition within a file, resides in the secondary-storage file space; a copy of the context structure for the presently addressed process definition is kept in primary storage. The context structure contains two kinds of information: 1) labels and their attributes that can be addressed throughout the labeled-process definition, and 2) the hierarchy of open-process definitions and instances of labeled processes within the labeled-process definition.

The label structure represents labels that are addressable throughout the labeled-process definition but not those (connector labels and exits from open processes) that are planar bound. The structure is built from label references appearing in flowcharts and code planes and from those written explicitly on the process-definition data form, which is constructed from this label structure. An object ring exists for each label; it specifies: 1) the label, 2) its type (e.g., automatic, formal, etc.), 3) a set of description objects (one for each line on the data form associated with the label, plus the pseudo operation code, data declaration, and comments), 4) a responsibility set denoting the plane-structure data set ID for the plane in which the label was referenced, and 5) a count of the number of references in that plane. The description objects, which are variable in length, are a function of the length of the commentary printed by the user. The three keepers are used as positional markers as the user edits a statement or graphical symbol that references a label.

The process name (context label in Fig. 11) and its internal identifier also appear on the structure.

The remainder of the structure specifies the hierarchy of open planes in the labeled-process definition and instances of other labeled processes (virtual contexts) in

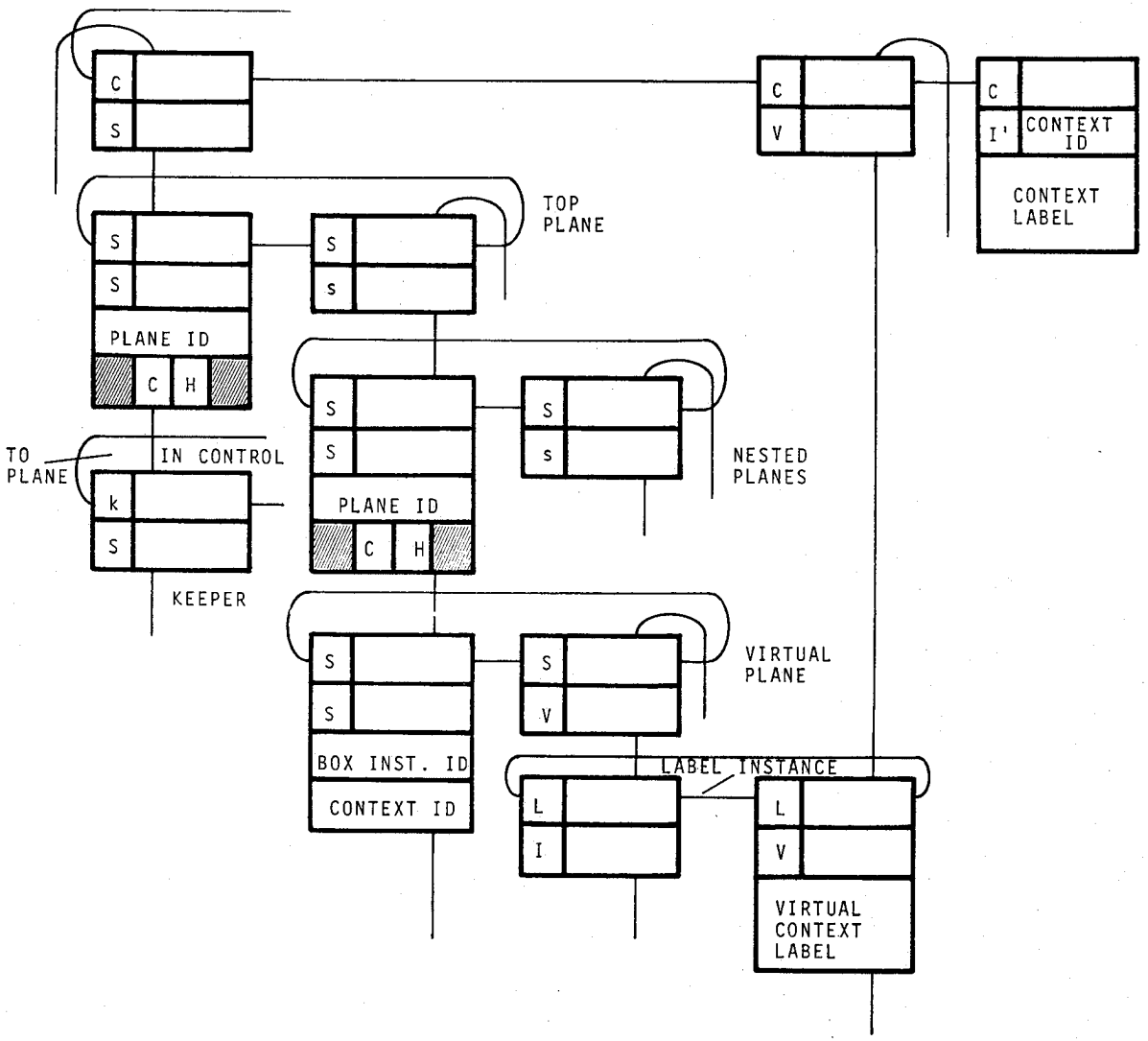


Fig. 11-Context Structure

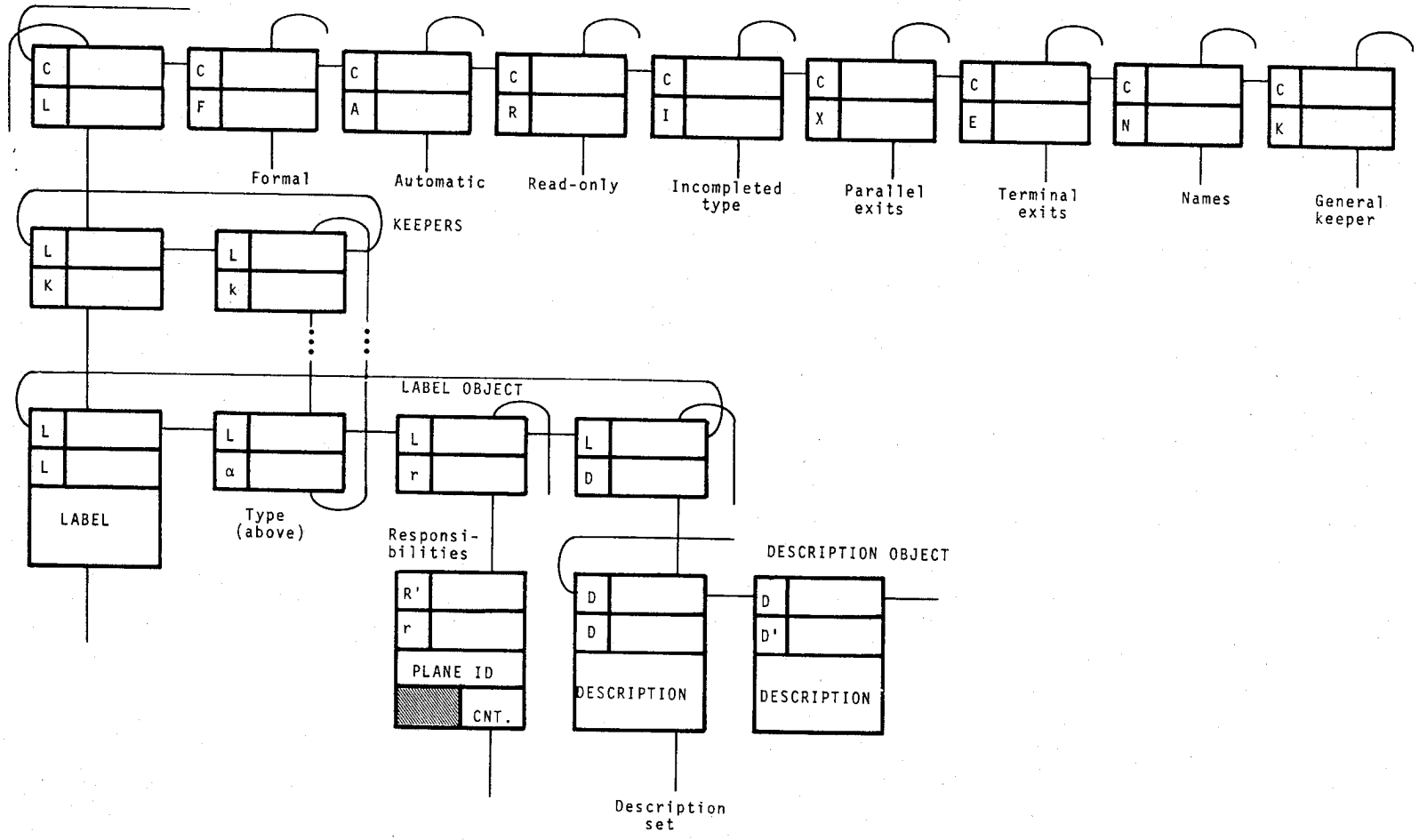


Fig. 12--Context Structure

the process definitions. In Fig. 11 (p. 36), each S/s set represents the position of a plane; the S/S are instances in that plane. If they object-connect to an S/s, an open-process definition is referenced; and if to an S/V, an instance of a labeled process is referenced. The plane-structure ID and the relative location (to the file base) are given for open-process definitions (planes) in the labeled-process definition. Other labeled-process definitions are virtual within this labeled definition so that their instances give the virtual-process name, its context structure ID (the corresponding location C, H is found in this plane structure), and the ID of the labeled instance in this plane structure. The keeper on the C/S set is object-connected to the particular plane whose frame is being displayed; it is used to mark the position, for example, for the return button function; i.e., going up one level in the structure to the parent plane.

PLANE STRUCTURE

There is at least one plane structure in the secondary-storage file space (Figs. 13-18) for each process definition. A copy of the plane representing the process being operated upon is in primary. The plane structure contains information about: 1) all frames (displays) within the plane, and 2) the graphical-element connectivity and label translation for each frame.

Figure 13 shows the basic object ring containing the various attribute sets and the plane-structure data set ID.

Figure 14 shows the frames structure for frames in the plane. Each frame object specifies the following:

1) The display frame ID and relative location to the file base;
2) A source of tag identifiers that become uniquely associated with each graphical element and that

are used with the hardware match circuitry to detect addressing of flow lines;

3) A set of CCW objects, one for each figure displayed in the frame.

The item element connects to a graphic object ring specifying the attributes of the displayed figure. For the frame being displayed, an external element (the PCCW) replaces the CCW displacement element on the CCW object.

Figure 15 shows the translation of formal parameters for a process instance and also the set of defined decisions.

Figures 16-18 show the graphic object rings and their attributes; e.g., flow connectivity.

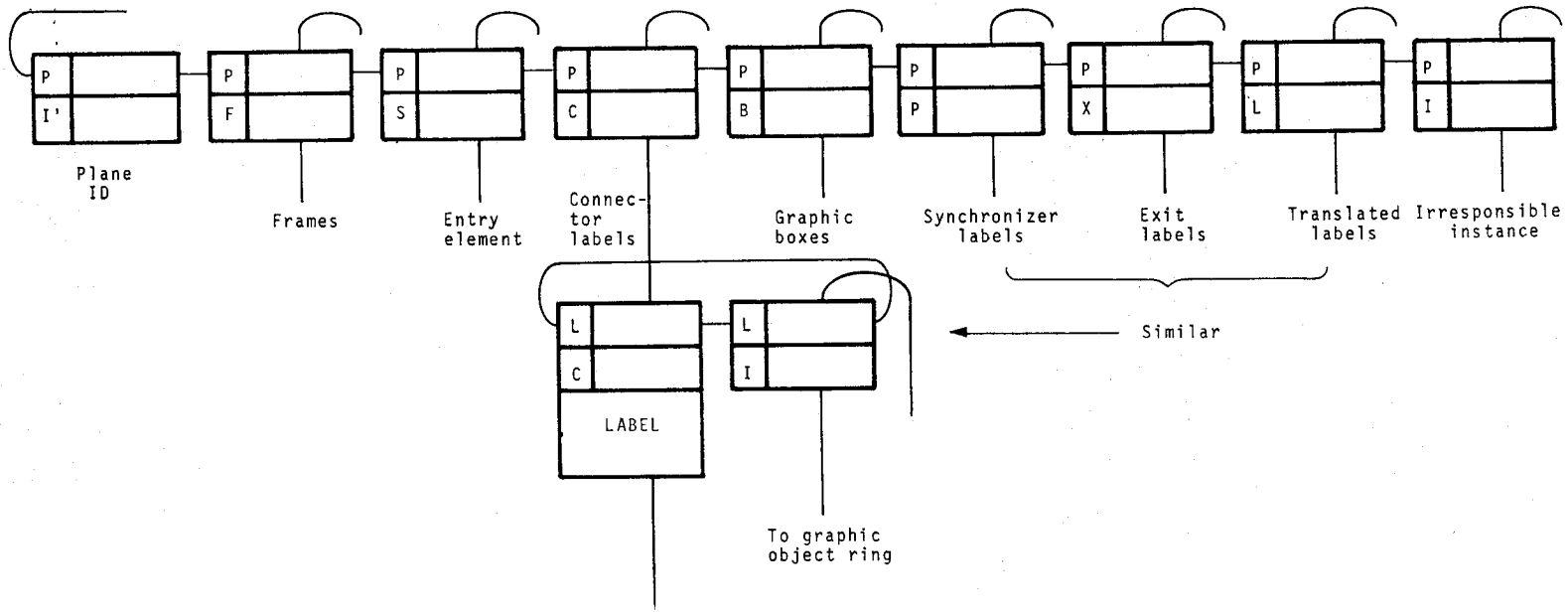


Fig. 13--Plane Structure

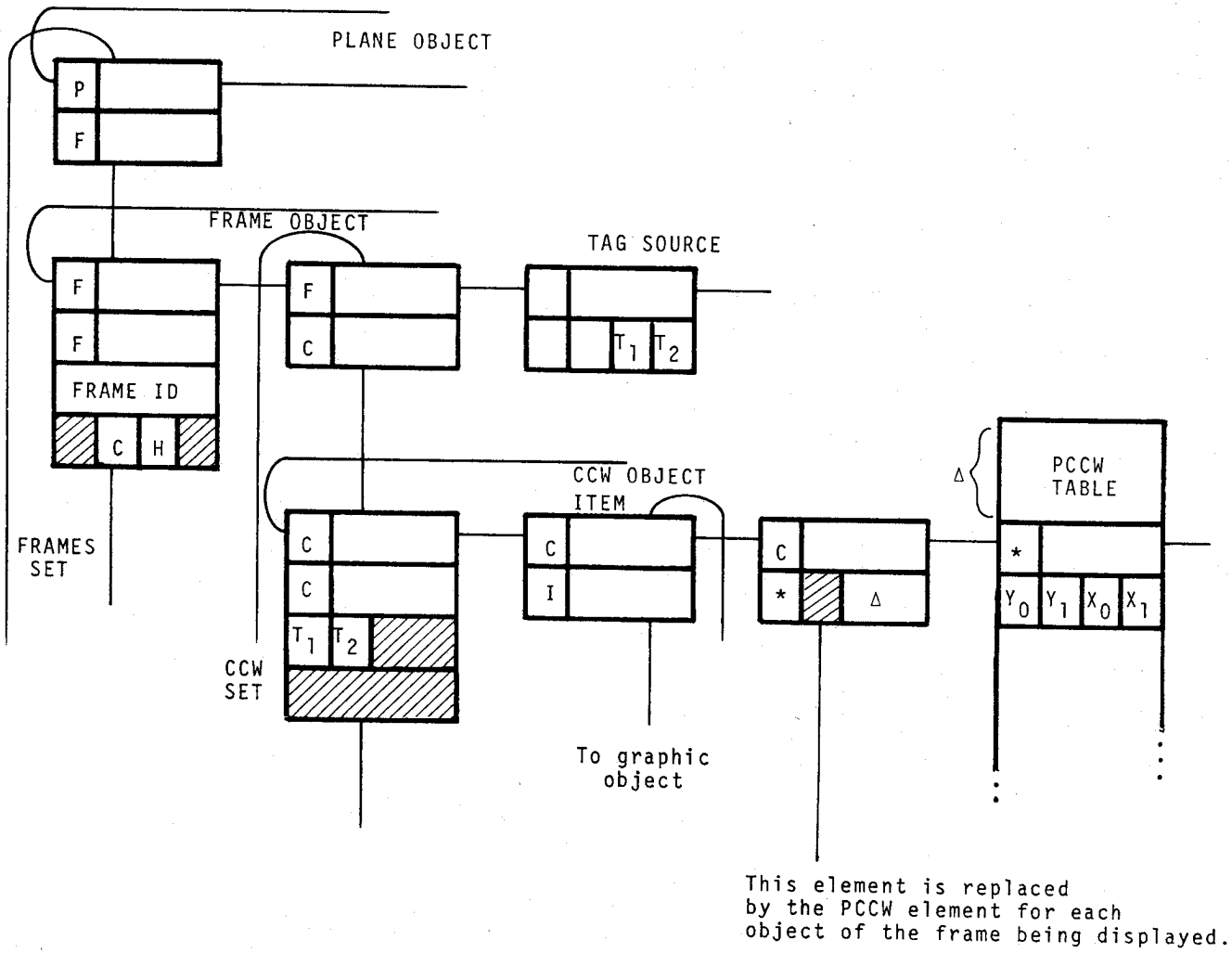


Fig. 14--Plane Structure

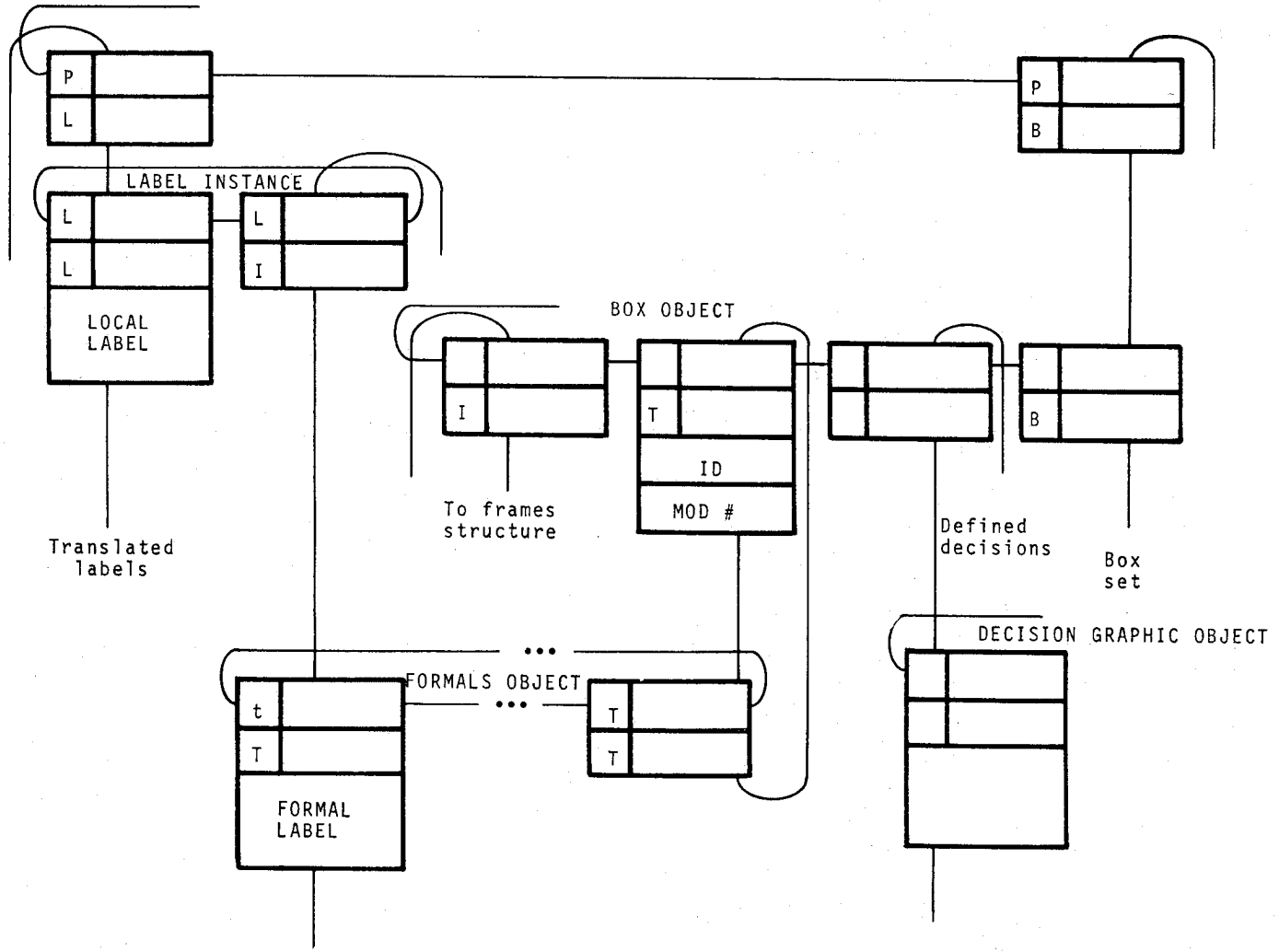


Fig. 15--Plane Structure

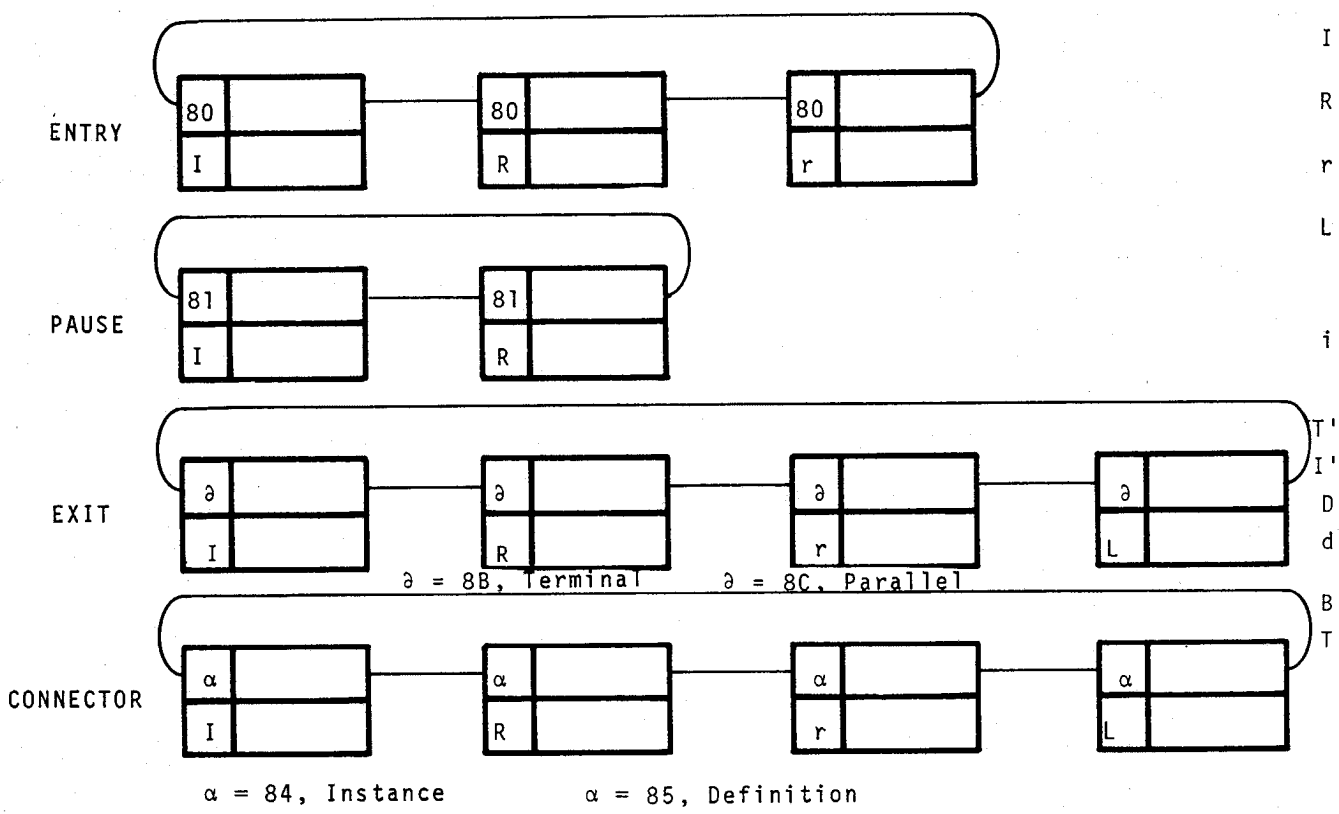


Fig. 16--Plane Structure

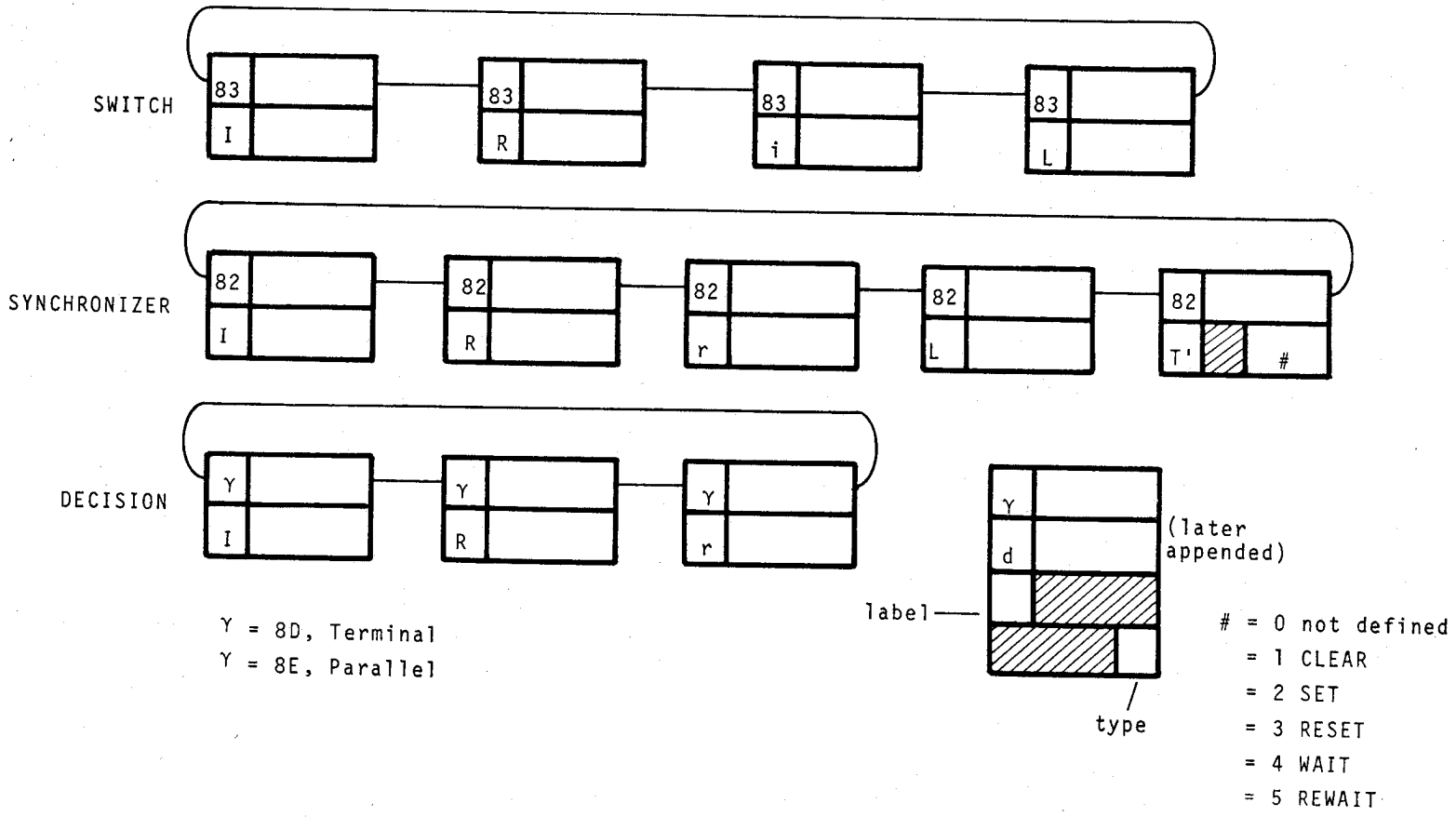
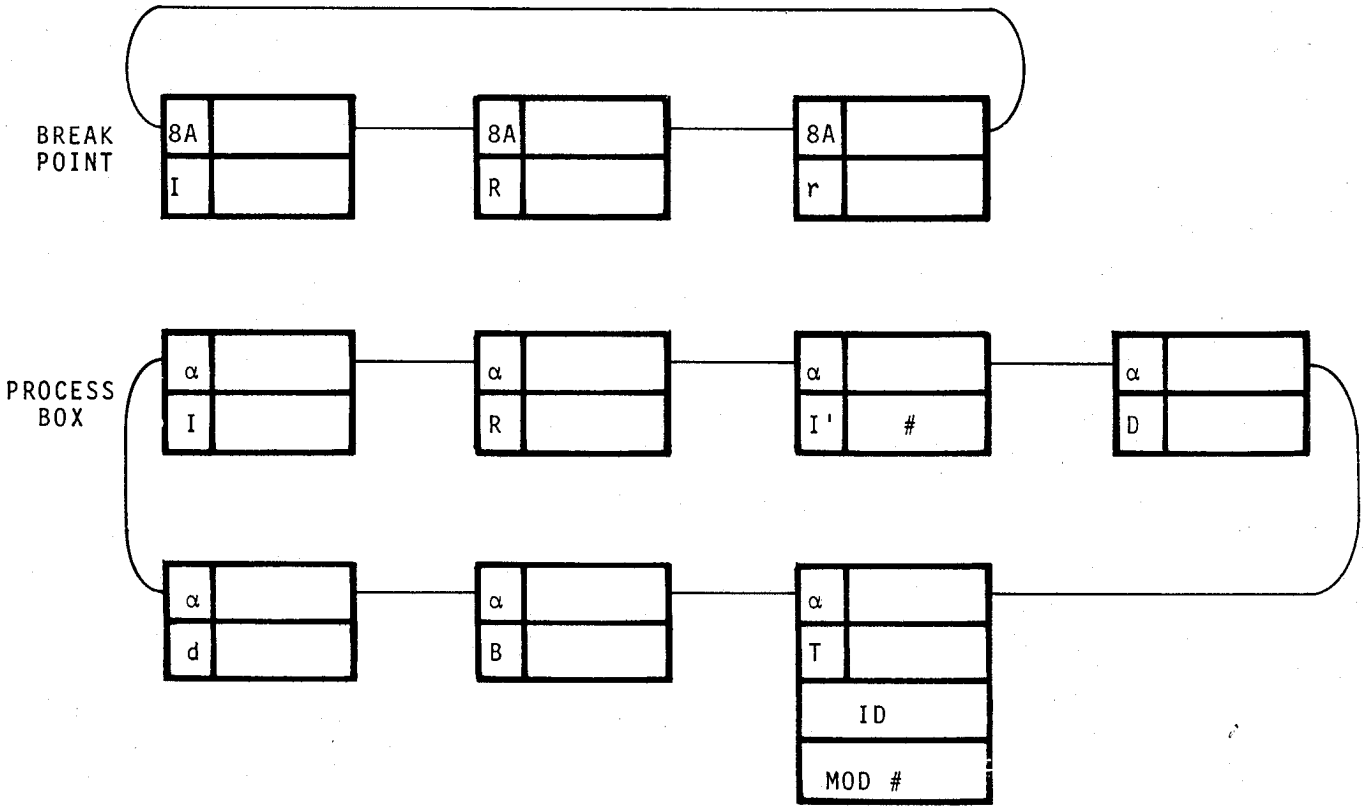


Fig. 17--Plane Structure



α = 86, Closed process
= 87, Open process
= 88, Remote code sequence
= 89, Serially reusable process
= 8F, Code box

Fig. 18--Plane Structure

Appendix B

LOADING AND THE CORE CONTROL DICTIONARY

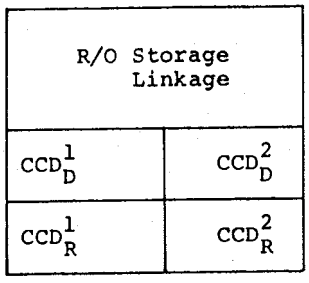
The format of process groups is similar to IBM link-edit modules. The assembler output is preprocessed before it becomes a secondary-storage† data set to eliminate, for example, the linkage table.

The processes are demand-loaded, and external references are resolved at load time, if appropriate. A control dictionary is maintained in primary storage† to permit such loading.

The Loader loads read-only process groups from secondary storage (upon request from the Supervisor) and allocates space for them. The request is issued when a process encounters an instance of another process that is not presently in primary. An SVC instruction trap occurs in the parent process for daughter process use; the Supervisor returns to the parent after loading. No explicit request for loading appears in the parent flow diagram.

The loader maintains a Core Control Dictionary (CCD), which is a directory of inter- and intra-process references used to resolve references between newly-loaded processes and those already loaded. The CCD is also used to absolve references when processes are released from primary.

Indexes to information concerning references for a process group are kept along with the process group in read-only storage as shown below.



†In the remainder of this discussion, "primary" and "secondary" will be used for "primary storage" and "secondary storage," respectively.

The indexes relate to the first and last entries of a definition and a reference section, respectively, as shown below.

$$CCD_D^i = \frac{\left|\left(\begin{array}{c}\text{ctl. dict. def. table}\\ \text{start address}\end{array}\right) - \left(\text{ctl. dict. def. tbl.}\right)\right|}{\text{the entry length}}$$

CCD_R is similarly computed for the reference section of the CCD.

The CCD indexes allow the loader to absolve references from other processes to a given process group when the group is released from core.

The R/O process-group data set on secondary is composed of the following four sections:

Definition Table (DT)
External Label Reference Table (LR)
Numbered Reference Table (NR)
Text (TXT)

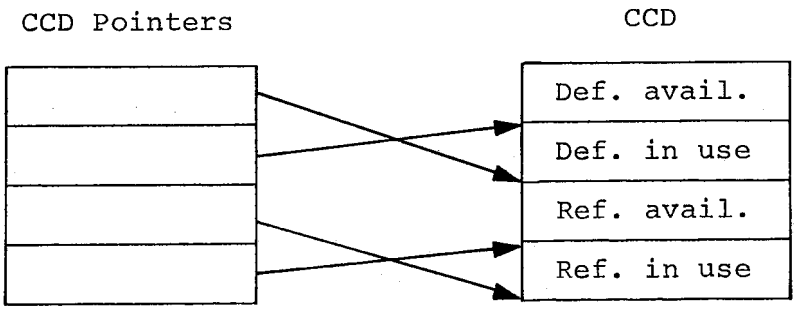
The R/O data sets originate from preprocessing assembler output.

DT is a transformation of the section-definition (SD) data on external-symbol-dictionary (ESD) images, which correspond to a control section for each process within the process group. LR entries are generated from a particular subset of the external-reference (ER) data on ESD images. LR is made up of those ER data that are external references

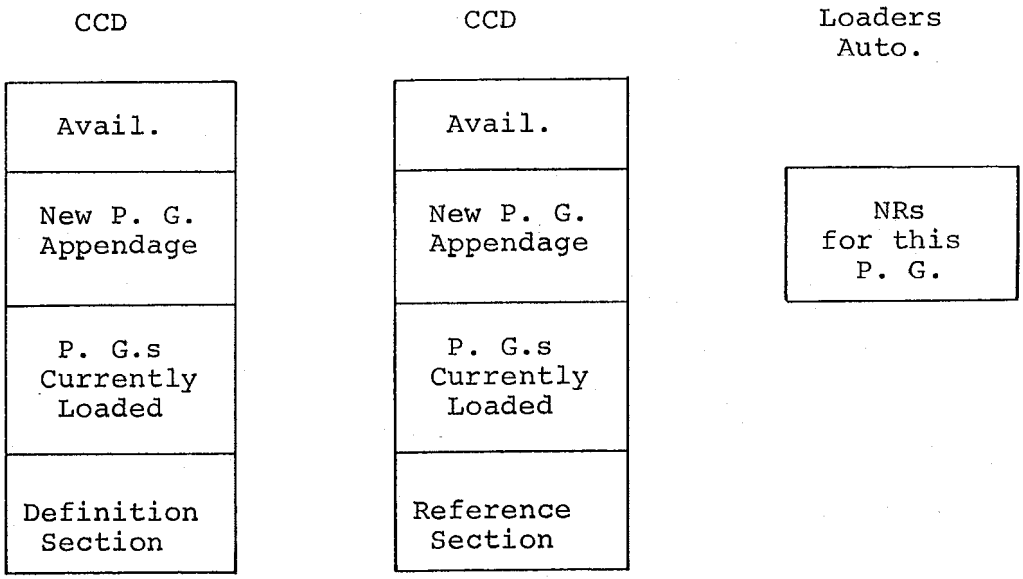
to the entire process group. NR is an abstraction of label definition (LD), label reference (LR), and the remainder of the ERs; i.e., those ERs whose reference is external to the control section (process) but is local to the module (process group). The TXT is the read-only code for all processes within the process group.

NRs are resolved quickly compared to LRs and need not be maintained in the CCD since they are local to the process group.

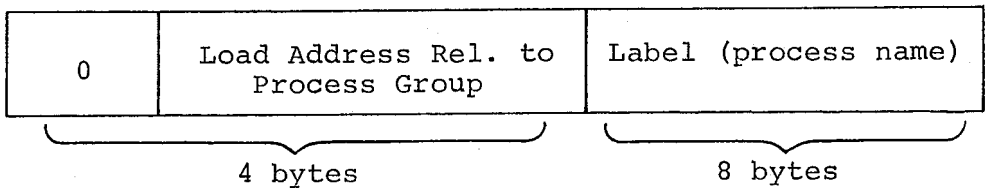
The CCD contains information for resolving references to a process definition (its read-only code) from a parent process instance at the time the instance is invoked or before. A set of four CCD pointers are kept to address the extremities of the table in use, as shown below.



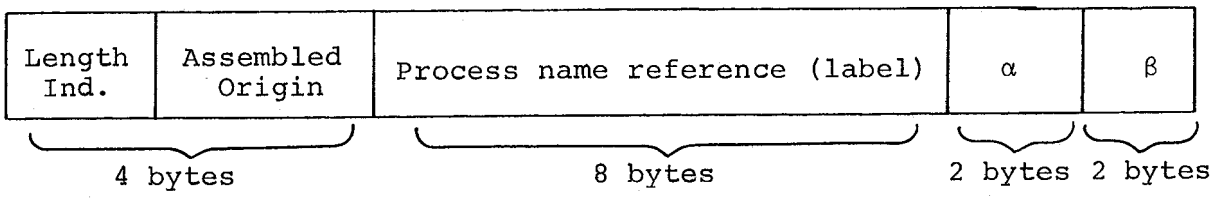
The DT and LR sections for the process group being loaded are appended to the CCD by a logical I/O process reading the information from secondary. The NR is read only into local automatic of the Loader since it is not a part of the CCD (see below).



A DT entry from the data set:

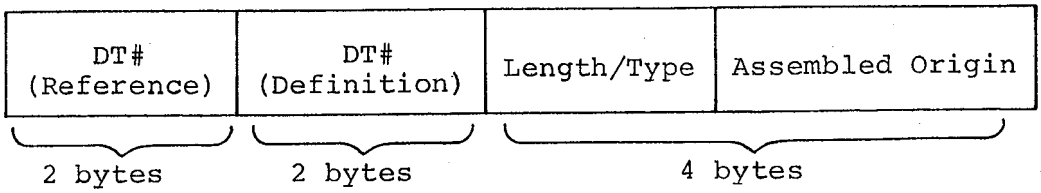


An LR entry:



α is resolved by the Loader to the relative definition table number for the reference, and β is the relative definition table number for the process definition containing the reference address constant.

An NR entry:



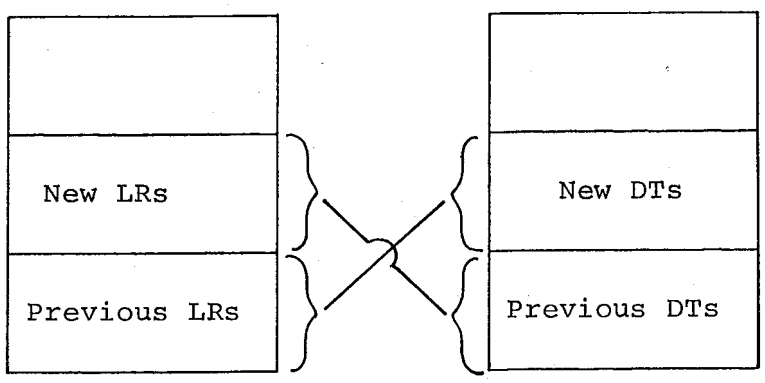
The Loader first updates the new appendage to the definition table by converting the relative-load addresses of the processes to absolute. This is done by adding the absolute-load address to the process group (determined by the loader when available space was found for the process group) to the relative address of the DT entry, which existed on the data-set disk image. The absolute address of the process replaces the relative address of the DT entry for the process.

Next, the modularly local references (NRs) are updated. The DT numbers are converted to reference the DT entries of the referencing process and the process being referenced, respectively. The assembled origin of the NR entry is really the displacement within the referencing process of the address constant. The referencing-process absolute address from the DT entry, plus the displacement, addresses the address constant. The address-constant length is supplied in the NR entry so that the constant may be extracted and added to the absolute address of the defining process (obtained from the DT entry of the process being referenced); this sum replaces the address constant.

The Loader resolves references between the newly-loaded process group and all other process groups currently loaded--this will include, at least, the parent process that has just invoked an instance of the new process being loaded. Normally, additional references are included as well.

The definition-table index of an LR entry in the data set on secondary is relative to the base of the DT section on secondary. The Loader computes the bias as a function of the displacement of the new DT section from the origin of the CCD's definition table.

The bias is added to the relative number as each LR for the new process group is resolved.

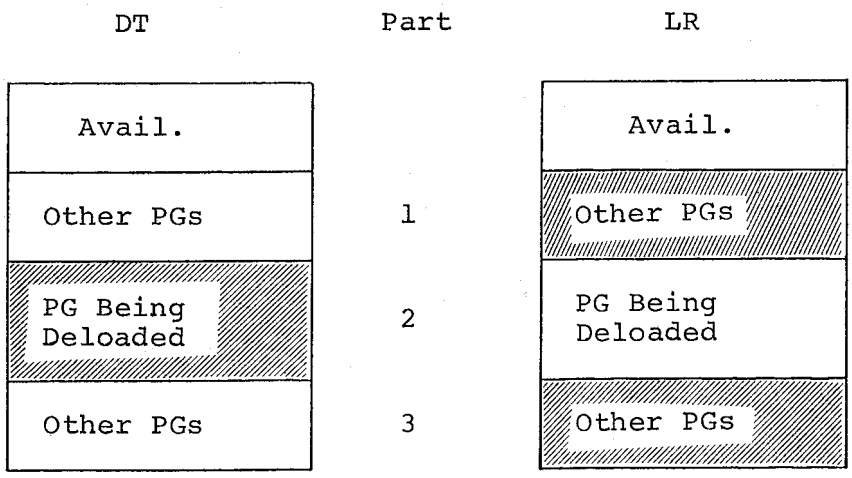


The Loader passes parts of the CCD twice in attempting to resolve LRs. The new LRs are matched successively against each previous DT entry in an attempt to resolve references in the newly loaded process group to processes already loaded. Another pass similarly matches the previous LRs with the new DTs in an attempt to resolve address constants between process groups. A match on the parent instance LR (which caused the loading of this new process group) will be found on this pass.

An LR check is as follows: If the entry has been neither deloaded nor resolved, the LR label is compared with successive labels of the corresponding DT subset. If a match is found, that DT entry index is inserted in the relative definition-table entry position of the LR entry; the entry is marked resolved, and the actual address constant within the read-only code is modified in a fashion similar to the NR computations described above.

The Loader inserts the DT and the LR boundary indexes for the new process group in the R/O linkage of the process group for later deloading.

The Loader deloads processes to provide additional space. The LR-DT processing for deloading is similar to loading for the purpose of locating references between the process group being released and others that are still in use. The actual address constants addressed in the match are modified at resolution time; i.e., the absolute-reference-process address is subtracted from the contents of the address constant. The LRs are then marked unresolved. The R/O linkage is adjusted to incorporate the deloaded process-group space into available space. The CCD is then compressed if possible; i.e., entries bounding the next available entry for either the DT or LR have been marked available. The available delimiters are repointed to include any available bounding entry space.



The deload match is between parts 1 and 3 of LR and part 2 of DT. Then part 2 of LR and part 2 of DT are marked available and possibly compressed into available space if part 1 has a zero length.

The NR section for the process group being deloaded does not exist and is of no consequence in deloading since all references were within the process group now released.

REFERENCES

1. Ellis, T. O., J. F. Heafner, and W. L. Sibley, *The GRAIL Project: An Experiment in Man-Machine Communications*, The RAND Corporation, RM-5999-ARPA, September 1969.
2. -----, *The GRAIL Language and Operations*, The RAND Corporation, RM-6001-ARPA, September 1969.
3. Bernstein, M. I., and H. L. Howell, *Hand-Printed Input for On-Line Systems: Final Report for Phase I*, System Development Corporation, TM-(L)-3964/000/00, Santa Monica, California, April 1968.
4. Davis, M. R., and T. O. Ellis, *The RAND Tablet: A Man-Machine Graphical Communication Device*, The RAND Corporation, RM-4122-ARPA, August 1964. (Also, *Proceedings of the FJCC, 1964*, p. 325.)
5. Ellis, T. O., and W. L. Sibley, *On the Development of Equitable Graphic I/O*, The RAND Corporation, P-3415, July 1966.
6. -----, *On the Problem of Directness in Computer Graphics*, The RAND Corporation, P-3697, March 1968.
7. Groner, G. F., *Real-Time Recognition of Handprinted Text*, The RAND Corporation, RM-5016-ARPA, October 1966. (Also, *Proceedings of the FJCC, 1966*, p. 591.)
8. Licklider, J.C.R., and W. E. Clark, "On-Line Man-Computer Communications," *Proceedings of the SJCC, 1962*, p. 113.
9. Sutherland, E. I., "Sketchpad: A Man-Machine Graphical Communication System," *Proceedings of the SJCC, 1963*, p. 329.
10. Sutherland, W., *Semiannual Technical Summary Graphics*, Lincoln Laboratory, Lexington, Mass., November 30, 1967, pp. 24-27.
11. Sutherland, E. I., and R. F. Sproul, "A Clipping Divider," *Proceedings of the FJCC, 1968*, pp. 765-775.